Hierarchical Multiprocessor CPU Reservations for the Linux Kernel

Fabio Checconi, Tommaso Cucinotta,
Dario Faggioli and Giuseppe Lipari

December 10, 2009

Goal

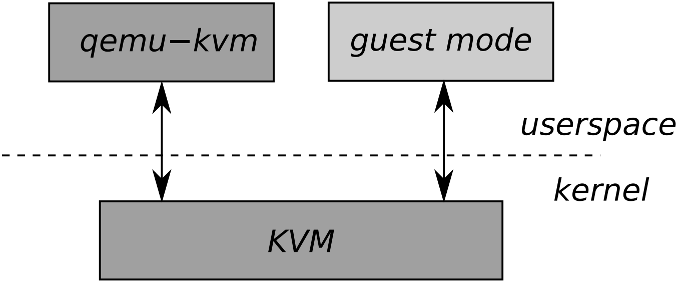
Support arbitrary CPU reservations in the Linux kernel, while preserving POSIX compliance and the current scheduler structure as much as possible.

CPU Scheduling in the IRMOS Project

IRMOS uses KVM to deploy its VMUs.

KVM is a userspace program from the scheduler's perspective.

KVM Architecture



Scheduling Requirements

CPU is Yet Another Resource, (on the host side) we need a scheduler:

- that can handle multiprocessor virtual machines (KVM is used to deploy VMs hosting services);
- that supports hard limits (people buy service time);
- that provides predictable response times (real-time services must respect real-time constraints).

Requirements Remapping

Almost everything is already there...

- each VM is put in its own cgroup;
- sched_rt and throttling expose an interface to support predictable service and hard limits.

Our paper describes how we enhanced throttling basing it on EDF and on a new system model/analysis recently introduced by Bini et al.

CGroup Interface

Common Grouping infrastructure:

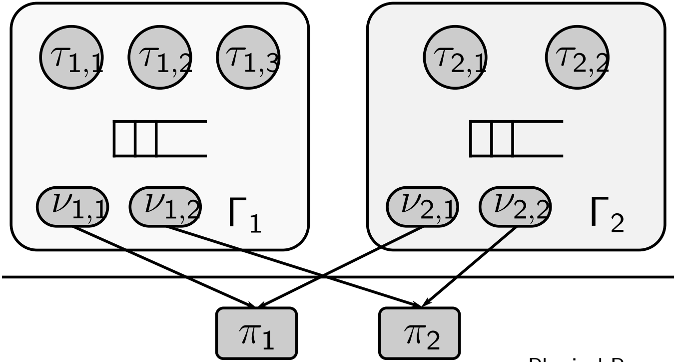
- each task belongs to a group (by default the root one).
- Groups are organized hierarchically.
- Each resource (CPU, network, disk etc.) has its own controller, granting access to tasks according to the group they belong to.

System Model

Most of what follows is borrowed from “The Multi Supply Function Abstraction for Multiprocessors,” by Bini et al., RTCSA ’09.

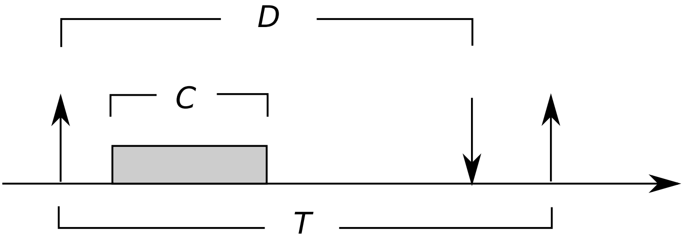
- Tasks belonging to the same application are grouped in the same task group;
- each task group receives service from a set of independent *virtual processors* $\nu_{i,1,\ldots,m}$;
- whenever a virtual processor is selected for execution, a task belonging to its task group is scheduled.

Block Diagram



Physical Processors

Task Model



- C—computation time
- D—deadline
- T—period (periodic)/minimum interarrival time (sporadic)

Servers

Servers are used to provide CPU reservations to tasks or to sets of tasks.

- Q—budget (how much execution time the server gets every P)
- P—period (how often the server gets its Q)

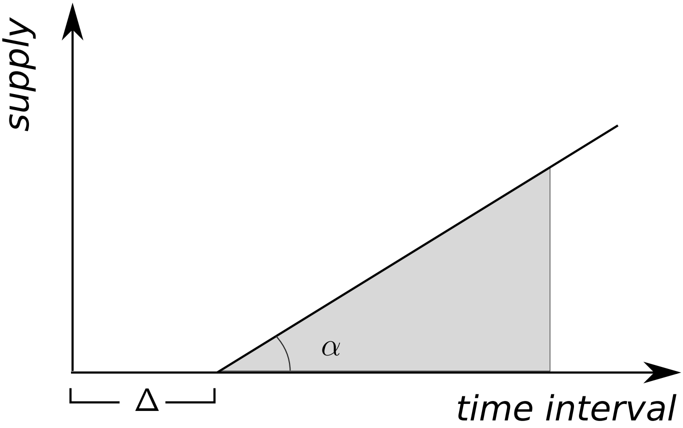
α, Δ

To characterize how each virtual processor receives service from the physical processors it is scheduled on, we use the (α, Δ) model, which characterizes the service in terms of *bandwidth* α, and *delay* Δ.

For the H-CBS server we're using, we have:

$$\alpha = \frac{Q}{P} \qquad \Delta = 2P - 2Q.$$

α, Δ (2)



Scheduling Algorithm

The system model allows for a number of possible configurations; we opted for:

- Partitioned, hierarchical Hard-CBS to schedule virtual processors on physical CPUs;
- Global fixed priority scheduling among tasks on the same task group;
- Static, symmetric bandwidth assignment among virtual processors: If a task group is assigned Q_i/P_i all of its virtual processors will get Q_i/P_i.

H-CBS

The Hard-CBS is a non workconserving scheduling algorithm based on EDF.

Each scheduled entity (virtual processors in our case) can be assigned a share of the physical processor time, in the form of Q time units every P. If an entity requires more than allocated it is throttled.

M(α, Δ)

Bini et al. introduced a way of composing multiple single CPU reservations into a single multiserver one.

Using their and other known results allows us to derive a schedulability test for our algorithm.

Interfering Workload

For each task τ_k we need to consider the interfering *workload* from higher priority tasks:

$$\overline{W}_k^{\mathrm{FP}} = \sum_{i=1}^{k-1} \overline{W}_{k,i},$$

where

$$\overline{W}_{k,i} = N_{k,i} C_i + \min\{C_i, D_k + D_i - C_i - N_{k,i} T_i\},$$

with $N_{k,i} = \left\lfloor \frac{D_k + D_i - C_i}{T_i} \right\rfloor$.

Interference

Now we can consider how the interfering workload is distributed among the various virtual processors, and find an upper bound to the interference:

$$\overline{I}_k = L_0 + \sum_{\ell=1}^{m} \min\left(L_\ell, \frac{\max\left(0, W_k - \sum_{p=1}^{\ell-1} pL_p\right)}{\ell} \right).$$

L_ℓ is the duration, in $[0, D_k)$, over which service is provided by ℓ virtual processors in parallel.

Schedulability

A task set $\Gamma = \{\tau_i\}_{i=1,\dots,n}$ is schedulable by a fixed priority algorithm on a set of virtual processors $\mathcal{V} = \{\nu_j\}_{j=1,\dots,m}$ modeled by $\{Z_j\}_{j=1,\dots,m}$, if

$$\forall k \in \mathbb{N} : 1 \leq k \leq n \qquad C_k + \overline{I}_k^{\mathsf{FP}} \leq D_k,$$

with $\{L_\ell\}_{\ell=0,\dots,m}$ calculated as follows:

$$\begin{array}{rcl} L_0 & = & D_k - Z_1(D_k) \\ L_\ell & = & Z_\ell(D_k) - Z_{\ell+1}(D_k) \\ L_m & = & Z_m(D_k). \end{array}$$

Scheduling in Linux

POSIX-like scheduling:

- 100 priority levels;
- several scheduling classes (`SCHED_RR`, `SCHED_FIFO`, `SCHED_OTHER`);
- strict priority service;
- `SCHED_RR` and `SCHED_FIFO` control how tasks with the same priority are handled;
- `SCHED_OTHER` have priority 0 (lowest) and are scheduled with CFS (a fair queueing variant).

The Scheduler

- One runqueue per CPU;
- priority arrays for RT tasks (one list per prio level, a bitmap to identify nonempty prio levels);
- a tree for CFS tasks;
- global enforcement of priorities on SMP;
- throttling of RT tasks.

Throttling Interface

To create a cgroup:

```
# mount -t cgroup -o cpu cgroup /dev/cgroup
# cd /dev/cgroup
# mkdir tg0
```

To limit its tasks to use no more than $Q = 20\text{ms}$ every $P = 100\text{ms}$:

```
# echo 100000 > tg0/cpu.rt_period_us
# echo 20000 > tg0/cpu.rt_runtime_us
```

Throttling vs. Server Scheduling

- Almost the same interface;
- throttling only *limits* the CPU time consumed by tasks, it does not *enforce* its provisioning (except in corner cases).

Implementation

- Use an RB tree to store groups, ordered by priorities *or* deadlines (boosting can promote a group to a fixed priority);
- changed the rt_bandwidth timer to be per-runqueue;
- added a *task runqueue* per each task group, to store its child tasks, which cannot be stored together with child runqueues (they have no deadline).

Tree Sorting

```
static inline int rt_rq_before(struct rt_rq *a,
       struct rt_rq *b)
{
        if ((a->rt_nr_boosted && b->rt_nr_boosted) ||
            global_rt_runtime() == RUNTIME_INF)
                return rt_rq_prio(a) < rt_rq_prio(b);
        if (a->rt_nr_boosted)
                return 1;
        if (b->rt_nr_boosted)
                return 0;
        return a->rt_deadline - b->rt_deadline < 0;
}
```

Task Runqueues

The only user-visible change is the introduction of task runqueues, needed to keep tasks separed from groups (groups have priorities only when boosted).

In addition to specify a Q/P assignment for each cgroup, the user has to specify an assignment for its task runqueues.

The bandwidth used for task runqueues cannot be used for groups.

Interface Implications

To create a task group, as usual:

```
# mount -t cgroup cgroup /dev/cgroup
# cd /dev/cgroup
# mkdir tg0
```

To assign $Q = 20\text{ms}$ over $P = 100\text{ms}$ to its tasks:

```
# echo 100000 > tg0/cpu.rt_period_us
# echo 20000 > tg0/cpu.rt_runtime_us
# echo 100000 > tg0/cpu.rt_task_period_us
# echo 20000 > tg0/cpu.rt_task_runtime_us
```

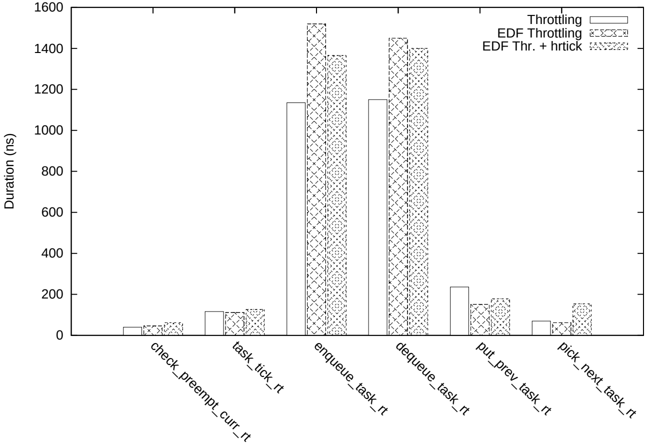
Data Structures

```
struct rt_edf_tree {
        struct rb_root rb_root;
        struct rb_node rb_leftmost;
};

struct rt_rq {
        struct prio_array active;
        u64 rt_deadline;
        struct hrtimer rt_period_timer;
        /* ... */
};
```

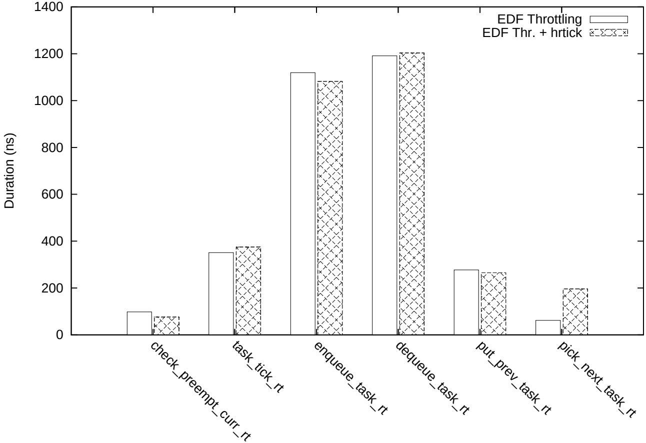
Data Structures (2)

```
struct task_rt_group {
        struct rt_rq **rt_rq;

        struct rt_bandwidth rt_bandwidth;
        struct task_group *tg;
};

struct task_group {
        struct task_rt_group rt_rq_group;
        struct task_rt_group rt_task_group;
        /* ... */
};
```

Overheads


Duration (ns)
0
200
400
600
800
1000
1200
1400
1600
Throttling
EDF Throttling
EDF Thr. + hrtick
check_preempt_curr_rt
task_tick_rt
enqueue_task_rt
dequeue_task_rt
put_prev_task_rt
pick_next_task_rt

HRTick



Duration (ns)
1400
1200
1000
800
600
400
200
0
EDF Throttling
EDF Thr. + hrtick
check_preempt_curr_rt
task_tick_rt
enqueue_task_rt
dequeue_task_rt
put_prev_task_rt
pick_next_task_rt

Future Work

From an academic POV:

- Give a formal treatment to shared resources access;
- evaluate bandwidth partitioning alternatives.

About the code:

- evaluate overheads more extensively;
- one cpupri per task group;
- auto-determined bandwidth for task runqueues;
- and many, many others...